Slime Smash Showdown: protótipo de um jogo *survivor-rogue-like* desenvolvido utilizando a Godot Engine

Gabriel da Silva Alves[1], **Isaias Mendes de Oliveira**[1]

[1]Instituto Federal de Educação, Ciência e Tecnologia de São Paulo (IFSP)
Campus Hortolândia – Vila São Pedro – 13.183-250 – Hortolândia – SP – Brasil

gabriel.alves2@aluno.ifsp.edu.br, isaiasmendes@ifsp.edu.br

Abstract. *The video game market has been growing exponentially, driven by free and open-source engines that enable independent developers to create innovative games. However, there is a lack of games implementing survivor-rogue-like mechanics in these engines. This study develops a 2D survivor game prototype with rogue-like elements using the Godot engine. The methodology includes creating a GDD (Game Design Document) to structure the project, followed by the iterative implementation of core mechanics such as movement, combat, and procedural progression. The prototype validated the feasibility of using Godot for this genre and demonstrated the effectiveness of a modular, component-based approach, allowing for scalability and efficient maintenance.*

Resumo. *O mercado de jogos eletrônicos cresce exponencialmente, impulsionado por engines gratuitas e open-source, permitindo a criação de jogos inovadores por desenvolvedores independentes. Contudo, há uma carência de jogos que implementem mecânicas survivor-rogue-like nessas engines. Este trabalho desenvolve um protótipo de jogo survivor em 2D com elementos rogue-like na engine Godot. A metodologia inclui a criação de um GDD (Game Design Document) e a implementação iterativa de mecânicas como movimentação, combate e progressão procedural. O protótipo validou a viabilidade da Godot para o gênero e demonstrou a eficácia de uma abordagem modular baseada em componentes, permitindo escalabilidade e manutenção eficiente.*

1. Introdução

O avanço de ferramentas como a *engine* gratuita **Godot Engine** (https://godotengine.org/) e plataformas de distribuição digital como **Steam** (https://store.steampowered.com/) e **Itch.io** (https://itch.io/) possibilita que desenvolvedores independentes criem jogos inovadores e alcancem um público global. Segundo [Newzoo 2021], a indústria de jogos movimentou 175,8 bilhões de dólares, com previsão de crescimento para 200 bilhões até 2024.

Essa democratização favorece gêneros como *survivor-rogue-like*, caracterizados por geração procedural e alta dificuldade, populares com títulos como Vampire Survivors [Galante 2022]. Esses jogos combinam simplicidade mecânica com profundidade estratégica, tornando-se relevantes para estudos de *design* e programação de jogos do estilo *suvivors*. Jogos como Hades [Supergiant Games 2020], Dead Cells [Motion Twin 2018] e Enter the Gungeon [Devolver Digital 2016] expandiram essas mecânicas, enquanto Vampire Survivors demonstrou a viabilidade do formato *survivor*. Contudo, a maioria dos títulos usa *engines* proprietárias, deixando alternativas *open-source*, como a Godot, em segundo plano.

A Godot Engine tem ganhado destaque, com crescimento expressivo na Steam e Itch.io entre 2022 e 2023 [Holfeld 2023]. Sua natureza *open-source*, facilidade de uso e suporte a jogos 2D e 3D a tornam uma ferramenta versátil. No entanto, há poucas referências sobre a implementação de mecânicas *survivor-rogue-like* nessa *engine*, dificultando sua adoção estruturada, especialmente para iniciantes.

Inspirado por Vampire Survivors, este projeto desenvolve um protótipo de jogo *survivor* 2D com mecânicas *rogue-like* na Godot. O jogador enfrentará hordas[1] de inimigos, evoluindo

[1]Um grande grupo, como uma multidão, massa.

ao derrotá-los e desbloqueando novas habilidades. O GDD (Documento do Projeto do Jogo, do inglês *Game Design Document*), disponível no Apêndice A, documenta as mecânicas, personagens e progressão. Os resultados indicam que a abordagem modular da Godot permitiu escalabilidade e fácil manutenção, viabilizando jogos do gênero *survivor-rogue-like*.

Este artigo organiza-se em seis seções: a seção 1 apresenta a introdução; a seção 2 traz o referencial teórico; a seção 3 discute trabalhos correlatos; a seção 4 descreve a metodologia; a seção 5 aborda o desenvolvimento e os resultados; e a seção 6 apresenta as conclusões e propostas futuras.

2. Referencial teórico

Nesta seção, são apresentados conceitos e estudos relacionados aos jogos digitais e seu desenvolvimento. O objetivo é fornecer uma base teórica para o projeto, evidenciando a relevância acadêmica e técnica das decisões tomadas ao longo da criação do protótipo.

2.1. O que define um jogo?

Para [Lucchese and Ribeiro 2012], os jogos assumem diferentes formas, como esportes, jogos de tabuleiro, videogames e brincadeiras, cada um com suas próprias regras e dinâmicas. Segundo [Petry 2020] definir o que é um jogo é uma tarefa complexa devido à sua diversidade, multifuncionalidade e natureza interdisciplinar. Além disso, a percepção do que constitui um jogo pode variar amplamente entre diferentes culturas e indivíduos, tornando desafiadora a formulação de uma definição universal.

[Huizinga 2019] descreve o jogo como uma atividade livre e voluntária, que se distingue da vida cotidiana por ocorrer em um espaço e tempo próprios, com regras estabelecidas e aceitas pelos participantes. Mais do que um meio de entretenimento, o jogo é uma expressão cultural que estimula a criatividade, a socialização e a construção de vínculos entre os jogadores. Sua capacidade de proporcionar prazer e envolvimento o torna uma forma essencial de interação humana, contribuindo para a formação da sociedade e da civilização.

[Salen and Zimmerman 2012] definem o jogo como um sistema no qual os jogadores participam de um conflito artificial, regido por regras, que resulta em um desfecho quantificável.

Neste contexto, [Petry 2020] identifica quatro características ontológicas dos jogos digitais: **(1)** a liberdade, na qual o jogador escolhe voluntariamente entrar no universo do jogo; **(2)** a produção de um estado de ânimo variável, que influencia a imersão e a experiência do jogador; **(3)** a presença de regras, que orientam o comportamento dentro do jogo, podendo ser seguidas ou modificadas pelo jogador; e **(4)** a transformação contínua dessas regras, que se adaptam conforme o jogo evolui.

Os jogos digitais se subdividem em diversos gêneros, que influenciam suas mecânicas e experiências. Entre eles, destacam-se os *rogue-like* e *bullet hell*, ambos relevantes para este trabalho. *Rogue-like* refere-se a jogos caracterizados por mapas gerados proceduralmente, morte permanente do personagem e alta rejogabilidade, exigindo que os jogadores se adaptem a cada nova tentativa. Já *bullet hell* é um subgênero dos jogos de tiro em que o jogador deve desviar de uma grande quantidade de projéteis na tela, exigindo reflexos rápidos e precisão.

2.2. Desenvolvimento de jogos digitais

O desenvolvimento de jogos digitais é um processo complexo e interdisciplinar que abrange diversas etapas, combinando arte, tecnologia e interatividade. De acordo com [Schell 2008], o processo de criar jogos é fundamentalmente interdisciplinar, reunindo profissionais de áreas como programação, *design* gráfico, narrativa e *design* de áudio.

[Fullerton 2008] descreve o desenvolvimento de jogos como um ciclo composto por cinco fases principais: concepção, pré-produção, produção, testes e lançamento. Esse ciclo permite que o projeto evolua de uma ideia inicial para um produto final jogável e de qualidade.

Um dos elementos fundamentais nesse processo é o GDD, que atua como um guia para o desenvolvedor, organizando todas as informações essenciais para a criação do jogo e garantindo

a coerência do projeto. Conforme [Schell 2008], o GDD é um documento vivo, que está em constante mudança à medida que o desenvolvimento do jogo avança. Ele registra as decisões e direções criativas do *game design*, servindo como referência para todos os envolvidos no projeto. Não existe um modelo ideal de GDD que sirva para todos os projetos, para cada jogo terá de ser avaliado um modelo que mais se ajuste. Por exemplo, para um jogo simples e com menos foco na narrativa, um modelo mais visual pode ser melhor, já um RPG[2] (Jogo Narrativo, do inglês *Role-Playing Game*) pode precisar de uma documentação extensa sobre a trama, personagens, ambiente etc.

No desenvolvimento de jogos, *assets* referem-se a qualquer recurso utilizado no desenvolvimento de um jogo, como modelos 3D, texturas, efeitos sonoros e música. Esses elementos são criados por profissionais diversos, como artistas, animadores e designers de som, e são essenciais para compor o mundo do jogo. Dependendo do estilo, gênero e escopo do projeto, diferentes tipos de *assets* são necessários. Por exemplo, *assets* visuais incluem personagens, objetos e cenários, enquanto *assets* de áudio envolvem efeitos sonoros e trilhas sonoras. Esses recursos são importados para o motor do jogo, onde são usados para dar vida ao universo virtual [Kalinin 2023].

2.3. Avaliação de jogos digitais

A avaliação de jogos digitais é um processo importante no ciclo de desenvolvimento, para que a experiência do jogador atenda às expectativas de qualidade, usabilidade e engajamento. Diversos modelos e abordagens são utilizados para essa finalidade, incluindo testes de jogabilidade, avaliações heurísticas e escalas de experiência do usuário.

Um dos modelos amplamente estudados é o **EGameFlow**, que propõe uma escala de autoavaliação para medir a experiência do jogador em relação a sete dimensões principais: concentração, clareza de objetivos, *feedback*, desafio, autonomia, imersão e interação social [Shu-Hui et al. 2018]. Essas dimensões são fundamentais para avaliar o envolvimento e a satisfação dos jogadores durante a interação com o jogo.

Além disso, testes práticos, como sessões de *playtesting*[3], são realizados para coletar *feedback* dos jogadores e refinar elementos de jogabilidade, *interface* e mecânicas [Comando Geek 2024]. A aplicação desses métodos contribui para um desenvolvimento mais iterativo e centrado no usuário, resultando em produtos finais de maior qualidade.

3. Trabalhos correlatos

Nesta seção, são apresentados os jogos que serviram de inspiração para o desenvolvimento deste projeto. Esses títulos possuem características relevantes tanto em termos de mecânicas quanto de *design* de jogo. A análise dessas referências permite identificar soluções já estabelecidas no gênero *survivor-rogue-like*, além de fundamentar as escolhas técnicas e criativas adotadas no desenvolvimento deste protótipo.

3.1. Vampire Survivors

Desenvolvido por Luca Galante e lançado em 2021, Vampire Survivors contribuiu significativamente para a popularização e evolução do gênero *survivor-rogue-like*. O jogo tem uma estética de horror gótico e combina elementos de *rogue-like* e *bullet hell*, desafiando o jogador a sobreviver contra hordas de inimigos que se tornam progressivamente mais difíceis. Sua mecânica central envolve movimentação contínua pelo cenário, enquanto armas e habilidades são ativadas automaticamente para eliminar adversários [Galante 2022].

A dinâmica de enfrentar ondas crescentes de inimigos e adquirir novas habilidades ao longo da partida foi uma influência direta para este projeto, que adota mecânicas similares como base para sua jogabilidade.

[2]É um tipo de jogo em que os participantes interpretam personagens e criam histórias juntos. O jogo segue regras definidas, mas os jogadores podem improvisar. As decisões de cada um influenciam o rumo da história.

[3]Um processo em que os jogadores testam o jogo em sua fase de protótipo.

3.2. Hades

Lançado em 2018 pela Supergiant Games, Hades é um *rogue-like* de ação que se destaca por seu combate intenso e sistema de progressão procedural. A cada nova tentativa, inimigos, poderes e habilidades são reorganizados, incentivando o jogador a experimentar diferentes combinações estratégicas. O jogo tornou-se um marco no gênero ao introduzir a mecânica de "progressão baseada na morte", na qual o jogador fortalece seu personagem mesmo após falhas sucessivas [Supergiant Games 2020].

A história de Hades segue Zagreus, o filho do deus grego Hades, enquanto ele tenta escapar do submundo e se rebelar contra seu pai. Ao longo de suas tentativas, ele encontra diversos deuses do Olimpo, que lhe concedem poderes especiais para ajudá-lo em sua jornada. A cada morte, ele retorna ao palácio de Hades, onde a história se desenrola e novas interações com personagens familiares do panteão grego acontecem.

Esses elementos dialogam diretamente com os objetivos deste projeto, onde a sobrevivência do jogador dependerá da escolha estratégica de habilidades e do domínio do combate dinâmico.

3.3. Turnip Boy Commits Tax Evasion

Criado pelo estúdio Snoozy Kazoo, Turnip Boy Commits Tax Evasion foi lançado em 2020, é um jogo de ação e aventura com um estilo visual carismático e uma narrativa humorística. Embora não pertença ao gênero *rogue-like*, sua estética única, mecânicas de exploração de masmorras (*dungeons*) e resolução de *puzzles* oferecem referências valiosas para este projeto [Kazoo 2019].

A maneira como o jogo recompensa a exploração e constrói sua atmosfera influenciou diretamente aspectos do *design* visual e da ambientação do protótipo em desenvolvimento.

4. Metodologia

Segundo [Fullerton 2008], o desenvolvimento de jogos eletrônicos possui várias etapas, desde a idealização até a prototipação para validação das mecânicas, passando pelo desenvolvimento, testes e publicação. O desenvolvimento deste projeto foi realizado com uma abordagem em fases, permitindo o acompanhamento do progresso de forma organizada e iterativa.

4.1. Planejamento

Nesta fase, é criado o GDD, uma documentação detalhada que guiará todo o processo de criação do jogo. O GDD descreve a idealização do jogo, definindo o escopo, detalhando as principais mecânicas, enredo, *designs* de personagens, requisitos técnicos e estéticos, além da escolha das ferramentas que serão utilizadas.

4.2. Definição da *Game Engine*

Para facilitar o desenvolvimento de jogos, são utilizadas *game engines*, que fornecem um conjunto de ferramentas e códigos-base para que os desenvolvedores criem, testem e implementem jogos sem precisar programar as mecânicas, cálculos de física e reprodução de sons, comuns aos jogos. A *engine* escolhida é a Godot, uma das principais ferramentas *open-source* usadas por desenvolvedores independentes [L'Italien 2024].

4.3. Prototipagem

A partir da definição da *engine* e das mecânicas principais, é desenvolvido um protótipo inicial que servirá como base para a construção progressiva do jogo. Esse protótipo inclui funcionalidades básicas, como movimentação do personagem, geração procedural de inimigos e coleta de recursos. Diferente de um protótipo descartável, esse modelo será continuamente refinado e expandido ao longo do desenvolvimento, permitindo a validação gradual das ideias de *gameplay* [Costa 2019].

4.4. Implementação Iterativa

Com a validação das funcionalidades iniciais, o desenvolvimento prossegue de forma incremental, adicionando novas mecânicas e refinando as existentes com base nos testes e nos *feedbacks* coletados. Cada iteração do ciclo de desenvolvimento incorpora melhorias e ajustes, assegurando que as mudanças sejam testadas e validadas antes da próxima etapa. Esse processo contínuo permite a adaptação do jogo conforme as necessidades identificadas durante a produção.

4.5. Testes

Durante cada ciclo incremental, são realizados testes internos para avaliar o desempenho das mecânicas de combate e da geração procedural, identificando pontos de melhoria. Por fim, são feitos testes gerais para identificar problemas de jogabilidade, *bugs* e desequilíbrios. Os *feedbacks* recebidos foram utilizados para ajustar a dificuldade, progressão e distribuição de inimigos e itens. Nessa fase, é realizado também o polimento, com ajustes visuais, sonoros e de otimização de desempenho, para garantir uma experiência final satisfatória ao jogador.

5. Desenvolvimento

O desenvolvimento do protótipo seguiu uma abordagem estruturada baseada no GDD. Após a definição do escopo e das principais mecânicas, foi criada a prototipagem para validar movimentação, combate e geração de inimigos.

Em seguida, a implementação refinou esses elementos, adicionando habilidades, menus interativos e ajustes de dificuldade. Por fim, testes foram realizados para garantir estabilidade, desempenho e uma experiência de jogo equilibrada. A seguir, são detalhadas cada etapa desse processo.

5.1. Planejamento

Para estruturar o desenvolvimento do projeto, foi criado o GDD, um documento utilizado para organizar o escopo, detalhar as mecânicas e registrar as características gerais do jogo. Seguindo o modelo de GDD completo sugerido por [Rollings and Morris 2004], o GDD foi continuamente refinado ao longo do processo, incorporando novas ideias e ajustando elementos conforme necessário.

5.1.1. Definição do escopo

O primeiro passo na criação do GDD foi definir o escopo inicial do projeto. Para definir os detalhes do escopo, foi feito uma seção de *brainstorming*. O jogo recebeu o nome de **Slime Smash Showdown**, e o gênero escolhido foi *survivor-rogue-like*, caracterizado pela sobrevivência do jogador contra hordas crescentes de inimigos enquanto coleta habilidades e melhorias. A principal inspiração veio do jogo Vampire Survivors.

O projeto foi planejado para ser desenvolvido em um ambiente 2D, utilizando *pixel art* para criar um estilo visual simples e estilizado.

5.1.2. Enredo

Após definir as características gerais, foi elaborado o enredo do jogo. A história se passa em um mundo de fantasia onde os *slimes* estão sendo caçados até quase a extinção. O jogador assume o controle de um grupo de *slimes* sobreviventes. Com a ajuda de uma bruxa, eles lutam para sobreviver contra hordas de cavaleiros que os caçam. A ideia do enredo é inspirada em uma inversão de protagonismo, onde um monstro clássico dos jogos se torna o herói e os cavaleiros se tornam os inimigos.

5.1.3. Personagens

Jelly: Protagonista, é um *Slime* com planta na cabeça e revólver. Sua habilidade dispara uma bala maior, com perfuração infinita e dano crítico. O *design* combina fofura e atitude durona, inspirado em Rebecca (Cyberpunk: Edgerunners). Arte feita no Aseprite.

Codessa: Bruxa androide que experimenta com *Slimes*. Embora neutra no rosto, expressa emoções pelo chapéu mágico. Dá missões e recompensas ao jogador. Inspirada em Shinonome Nano (Nichijou) e Stocking Anarchy (Panty and Stocking with Garterbelt), mistura estética robótica e mágica.

Steelon: *Slime* robusto em armadura pesada, com espada. Sua habilidade é um ataque giratório. Inspirado em Greater Dog (Undertale) e no herói de Gato Roboto, une imponência e carisma. Arte em desenvolvimento.

Smashia: *Slime* humanoide feminina com manoplas. Ataca empurrando inimigos e, na habilidade especial, aplica sequência rápida de socos. Inspirada em Tinkaton (Pokémon) e Vi (League of Legends). Arte em desenvolvimento.

5.2. Prototipagem

Com as mecânicas e o enredo definidos, o passo seguinte foi validar as ideias por meio da prototipagem, utilizando a Godot Engine, ferramenta gratuita e *open-source* amplamente adotada por desenvolvedores independentes. Foram implementadas as mecânicas iniciais de movimentação, combate, experiência e geração procedural de inimigos, permitindo verificar a viabilidade e ajustes necessários antes da implementação completa.

5.2.1. Mapa

Antes das mecânicas básicas, foi necessário criar uma cena principal responsável por organizar as demais cenas do jogo. Na Godot Engine, uma cena (*scene*) é um conjunto de nós (*nodes*) organizados hierarquicamente para representar elementos do jogo, como personagens, componentes e menus [Linietsky et al. 2014].

A Figura 1 apresenta o menu de criação de nós na Godot. Os nós podem se especializar para tarefas específicas, como o nó **Node2D**, que herda da classe base *Node* e é projetado para objetos bidimensionais.

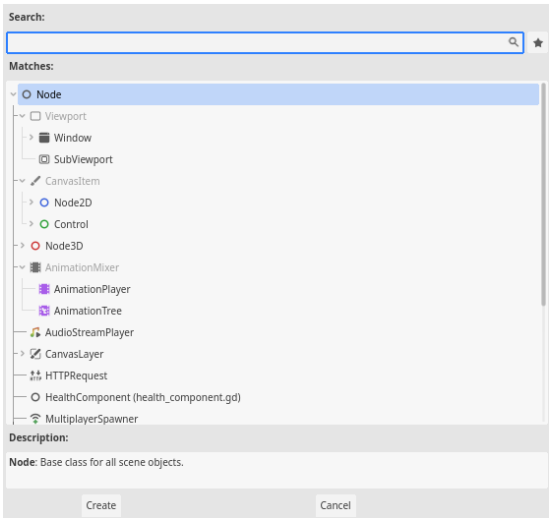

Figura 1. Menu de criação de *node*

Para criar o mapa do jogo, foi configurada a cena principal ***Main***, carregada primeiro na execução do projeto. Essa cena foi criada como um nó da classe *Node*, a mais genérica da

Godot. Dentro de ***Main***, foi adicionado um ***TileMap***, um nó especializado na construção de cenários baseados em mosaicos (*tiles*). O ***TileMap*** permite a criação eficiente de mapas ao organizar e renderizar blocos reutilizáveis, otimizando o desempenho do jogo. Esse nó utiliza um ***TileSet***, que contém as imagens dos elementos do cenário e possibilita a configuração de colisões, camadas e outras propriedades.

Na Figura 2, é apresentada a cena ***Main***, contendo o ***TileMap*** configurado. O ***TileMap*** fornece uma grade para desenhar o mapa com base nos *tiles* disponíveis. Para criar o ***TileMap*** e os elementos de *interface*, foram utilizados *assets* gratuitos disponívens no *site* https://kenney.nl/.

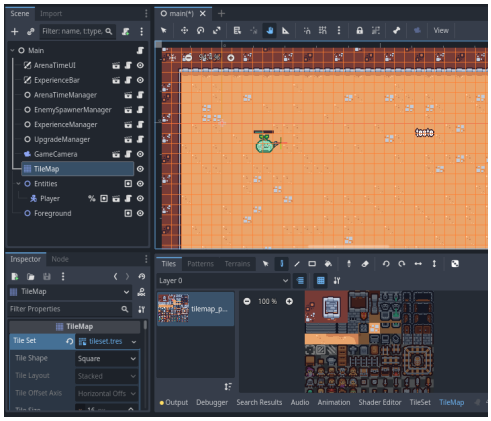

Figura 2. Cena *Main* com o *TileMap* selecionado

5.2.2. Jogador

Após a criação do mapa, foi desenvolvido um personagem jogável utilizando a classe ***CharacterBody2D***, na qual foram implementadas as mecânicas de movimentação, aquisição de habilidades, colisões e sistema de dano. Para tornar o projeto mais escalável e facilitar a manutenção, foi aplicado o padrão de composição, uma abordagem que consiste em construir funcionalidades a partir de pequenos componentes modulares e reutilizáveis.

A classe ***CharacterBody2D*** é especializada para o controle de personagens 2D, facilitando a implementação de movimentação, interação com o ambiente e física. A cena foi nomeada ***Player*** e contém diferentes áreas de colisão para funções específicas. Essas áreas são nós 2D configuráveis da Godot.:

- **Área verde**: Responsável pela coleta de recursos.
- **Área vermelha**: Detecta colisões com inimigos.
- **Área azul**: Define os limites de colisão com o terreno.

A Figura 3, ilustra essas áreas de colisão.

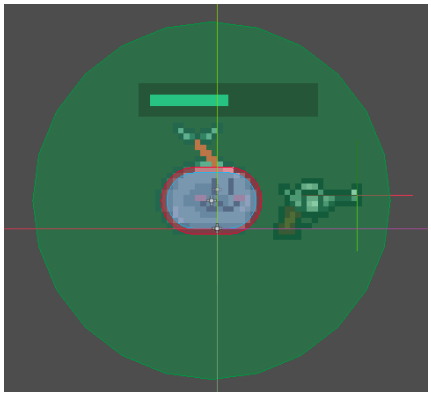

Figura 3. Colisões do *Player*

A movimentação do jogador é gerenciada pelo componente ***VelocityComponent***, que controla velocidade e aceleração. O *script* do *Player* recebe comandos, calcula a direção e atualiza a posição usando esse componente (veja na Figura 4).

```
45  func get_movement_vector():
46
47      var x_movement = Input.get_action_strength("mv_right") \
48      - Input.get_action_strength("mv_left")
49      var y_movement = Input.get_action_strength("mv_down") \
50      - Input.get_action_strength("mv_up")
51
52      return Vector2(x_movement,y_movement)
```

Figura 4. Cálculo da direção do *Player*

A função ***_process()*** executa a movimentação a cada quadro, normalizando o deslocamento com o parâmetro delta para garantir uniformidade. Ao se mover, o *script* ativa a animação *walk* e o *sprite* ajustado para a direção correta (veja na Figura 5).

```
29  func _process(delta):
30      var movement_vector = get_movement_vector()
31      var direction = movement_vector.normalized()
32      velocity_component.accelerate_in_direction(direction)
33      velocity_component.move(self)
34
35      if movement_vector.x != 0 || movement_vector.y != 0:
36          animation_player.play("walk")
37      else:
38          animation_player.play("RESET")
39
40      var move_sign = sign(movement_vector.x)
41      if move_sign != 0:
42          visuals.scale = Vector2(move_sign, 1)
```

Figura 5. Ajustes visuais para o andar do *Player*

Se um inimigo colidir com o jogador, um sinal contabiliza o dano recebido. A redução de HP é gerenciada pelo ***HealthComponent***, que emite o sinal ***health_changed***, capturado pelo *script* do *Player* e atualizado na *interface* por uma barra de progresso (veja na Figura 6).

```gdscript
func check_deal_damage():
    if number_colliding_bodies ==0 || !damage_interval_timer.is_stopped():
        return
    health_component.damage(1)
    damage_interval_timer.start()

func update_health_display():
    health_bar.value = health_component.get_health_percent()

func on_damage_interval_timer_timeout():
    check_deal_damage()

func on_health_decreased():
    GameEvents.emit_player_damaged()
    $HitRandomStreamPlayer.play_random()

func on_health_changed():
    update_health_display()
```

Figura 6. Lógica da vida do *Player*

A Figura 7 apresenta a hierarquia de nós do jogador, destacando as abas ***Scene*** e ***Script***, onde a função ***_ready()*** estabelece conexões entre sinais e funções.

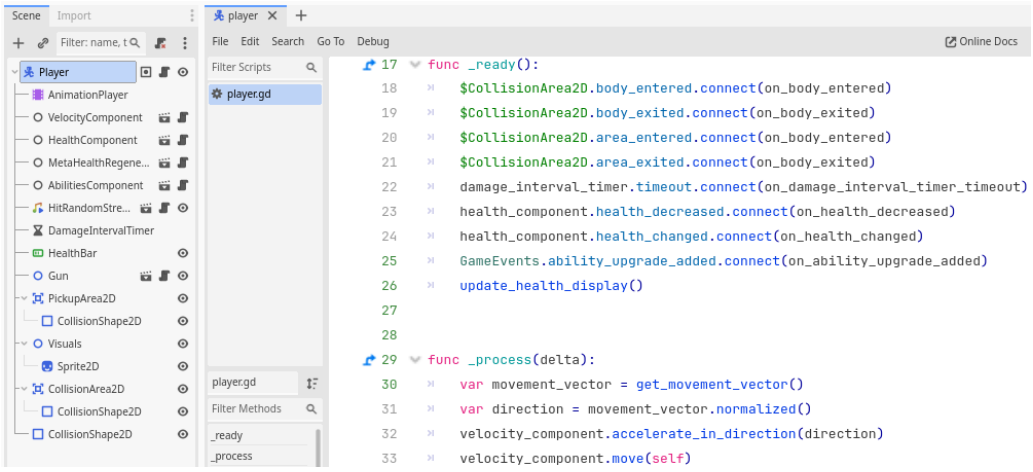

Figura 7. Cena do *Player*

5.2.3. Inimigo

Uma parte importante é o combate, e para isso foi necessária a criação de inimigos. O inimigo básico apenas persegue o jogador e causa dano ao contato.

Para facilitar a criação de diferentes tipos de inimigos no futuro, diversas funcionalidades foram modularizadas através da composição de componentes. A Figura 8, apresenta a cena ***BasicEnemy***, que define a estrutura base dos inimigos e os componentes que a compõem:

- ***HealthComponent***: Gerencia os pontos de vida do inimigo.
- ***VelocityComponent***: Controla o movimento do inimigo em direção ao jogador.
- ***HurtBoxComponent***: Detecta colisões e recebe dano.
- ***HitFlashComponent***: Exibe um efeito visual para indicar quando o inimigo é atingido.

- ***DeathComponent***: Padroniza o comportamento do inimigo ao ser derrotado.
- ***VialDropComponent***: Faz com que o inimigo derrube experiência quando eliminado, com base em uma probabilidade.

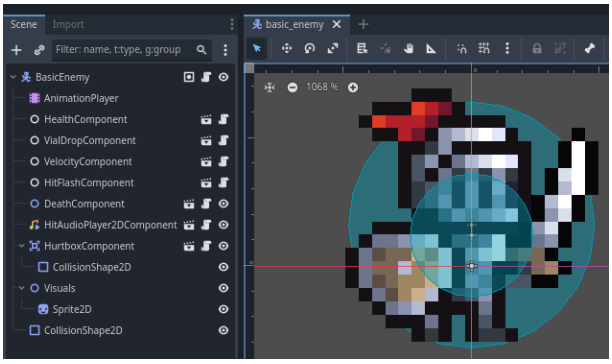

Figura 8. Cena *BasicEnemy*

O ***HurtBoxComponent***, ilustrado na Figura 9, é um componente para detecção de dano, responsável por registrar colisões com ataques (*hitboxes*) e aplicar o dano ao inimigo atingido. Ele é implementado como um nó do tipo ***Area2D***, podendo ser anexado a qualquer objeto que possa sofrer dano, como o jogador ou os inimigos.

```
14  func on_area_entered(other_area: Area2D):
15      if not other_area is HitboxComponent:
16          return
17
18      if health_component == null:
19          return
20
21      var hitbox_component = other_area as HitboxComponent
22      health_component.damage(hitbox_component.damage)
23
24      var floating_text = floating_text_scene.instantiate() as Node2D
25      get_tree().get_first_node_in_group("foreground_layer").add_child(floating_text)
26
27      floating_text.global_position = global_position + (Vector2.UP * 16)
28
29      var format_string = "%0.1f"
30      if round(hitbox_component.damage) == hitbox_component.damage:
31          format_string = "%0.0f"
32      floating_text.start(format_string % hitbox_component.damage)
33
34      hit.emit()
```

Figura 9. *Script Hurtbox*

Quando uma *hitbox* (área de ataque) entra na *hurtbox*, o evento de colisão é acionado pelo método *on_area_entered()*. O ***HealthComponent*** conectado à *hurtbox* reduz a vida do objeto atingido de acordo com o dano recebido na linha 22. Na linha 25, um número flutuante aparece sobre o inimigo, indicando o dano causado, ajudando o jogador a visualizar o impacto do ataque. Ao final, o componente emite o sinal "*hit*"na linha 34, para que outros sistemas, como efeitos visuais e sonoros, reajam ao evento de receber dano.

5.2.4. Gerenciador de criação de inimigos

Como o jogador enfrenta hordas constantes de inimigos, a instanciação desses inimigos precisa ser feita de forma eficiente e dinâmica, e, para isso, foi criado um gerenciador de criação de inimigos. Esse gerenciador funciona com um *timer*, que, ao atingir o tempo limite, instancia um inimigo ao redor do *player*. Conforme a dificuldade do jogo aumenta, o tempo de intervalo é reduzido e inimigos mais poderosos começam a aparecer. Isso é ilustrado na Figura 10.

```
45 func on_timer_timeout():
46     timer.start()
47 
48     var player = get_tree().get_first_node_in_group("player") as Node2D
49     if player == null:
50         return
51 
52     for i in number_to_spawn:
53         var enemy_scene = enemy_table.pick_item()
54         var enemy = enemy_scene.instantiate() as Node2D
55 
56         var entities_layer = get_tree().get_first_node_in_group("entities_layer")
57         entities_layer.add_child(enemy)
58         enemy.global_position = get_spawn_position()
```

Figura 10. ***Script EnemySpawnerManager***

Para o inimigo ser criado, o gerenciador busca a posição do *player* (linha 25) e gera uma certa quantidade de inimigos que varia dependendo do nível de dificuldade, um inimigo aleatório é escolhido (linhas 52 até 58) e instanciado em um ponto aleatório com base em um raio pré-definido ao seu redor (linha 32). Caso exista uma colisão entre o *player* (linha 36) e esse ponto, a posição de criação é rotacionada em 45° (linha 42), até conseguir uma posição válida. Isso serve para evitar que um inimigo seja criado fora do mapa e não consiga entrar na arena. Isso é ilustrado na Figura 11.

```
24 func get_spawn_position():
25     var player = get_tree().get_first_node_in_group("player") as Node2D
26     if player == null:
27         return Vector2.ZERO
28 
29     var spawn_position = Vector2.ZERO
30     var random_direction = Vector2.RIGHT.rotated(randf_range(0, TAU))
31     for i in 8:
32         spawn_position = player.global_position + (random_direction * SPAWN_RADIUS)
33         var additional_check_offset = random_direction * 20
34 
35         var query_parameters = PhysicsRayQueryParameters2D.\
36         create(player.global_position, spawn_position + additional_check_offset, 1)
37         var result = get_tree().root.world_2d.direct_space_state.intersect_ray(query_parameters)
38 
39         if result.is_empty():
40             break
41         else:
42             random_direction = random_direction.rotated(deg_to_rad(45))
43     return spawn_position
```

Figura 11. ***Script EnemySpawnerManager***

Após conseguir uma posição válida, um tipo de inimigo é escolhido aleatoriamente a partir de uma tabela ponderada, o inimigo selecionado é instanciado e posicionado na cena (veja a Figura 7). A tabela ponderada torna o projeto mais escalável, pois permite a adição de novos tipos de inimigos e ajustes dinâmicos na probabilidade de aparição de cada tipo de inimigo. Com isso, a distribuição dos inimigos pode ser balanceada conforme a progressão do jogo.

5.2.5. Arma

Para que o jogador possa enfrentar os inimigos, foi desenvolvida uma arma para o ***Player***. A cena da arma inclui um pivô central, ilustrado na Figura 12, permitindo que ela gire ao redor do jogador conforme a posição do cursor. Além disso, um segundo pivô, localizado na extremidade do *Sprite* da arma, define o ponto de origem dos disparos.

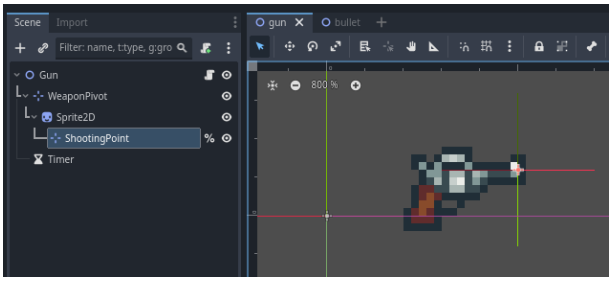

Figura 12. Cena Gun

A arma utiliza um temporizador para regular a frequência dos disparos. Na função ***_process()***, ilustrado na Figura 13, a posição do cursor do *mouse* é capturada, e a arma é rotacionada em sua direção, permitindo que o jogador mire livremente. Quando o temporizador é concluído, uma bala é instanciada na posição do pivô ***ShootingPoint***.

```
13  func _process(delta):
14      var mouse_position = get_global_mouse_position()
15      look_at(mouse_position)
16
17
18  func shoot():
19      var new_bullet = bullet.instantiate() as Node2D
20      new_bullet.global_position = shooting_point.global_position
21      new_bullet.rotation = self.rotation
22
23      var foreground = get_tree().get_first_node_in_group("foreground_layer") as Node2D
24      if foreground == null:
25          return
26      foreground.add_child(new_bullet)
27      new_bullet.hitbox_component.damage = damage
28
29
30  func on_timer_timeout():
31      shoot()
```

Figura 13. *Script Gun*

A bala é um nó ***Node2D*** que contém uma *HitBox*, responsável por interagir com a *HurtBox* do inimigo. Ao ser instanciada, ela registra a direção do disparo (linha 15) e se desloca

em linha reta (linha 16) até atingir um inimigo ou ultrapassar sua distância máxima (linha 19). Para tornar o combate mais dinâmico, a bala possui uma variável de perfuração(linha 6), permitindo que ela atravesse múltiplos inimigos antes de ser destruída. O projétil é removido da cena quando atinge o limite de perfuração(linha 24) ou percorre a distância máxima estabelecida (linha 19). Essas características estão ilustrados na Figura 14.

```
5   var travelled_distance = 0
6   var pierce = 0
7
8   func _ready():
9       $Area2D.body_entered.connect(on_entered)
10
11  func _physics_process(delta):
12      const SPD = 310
13      const RANGE = 1200
14
15      var direction = Vector2.RIGHT.rotated(rotation)
16      position += direction * SPD * delta
17
18      travelled_distance += SPD * delta
19      if travelled_distance > RANGE:
20          queue_free()
21
22  func on_entered(body):
23      pierce += 1
24      if pierce >= 3:
25          queue_free()
26
```

Figura 14. ***Script Bullet***

5.3. Implementação

Com as principais mecânicas em funcionamento, foi possível avançar para a implementação de novas funcionalidades e a correção de problemas de jogabilidade.

5.3.1. Habilidades

Um dos aspectos fundamentais que tornam jogos *rogue-like* divertidos é a variedade de habilidades adquiríveis ao longo da partida. A primeira habilidade desenvolvida foi a ***SwordAbility***, que gera uma espada próxima a um inimigo e o ataca automaticamente.

Essa habilidade utiliza um ***AnimationPlayer***, ilustrado na Figura 15, responsável por controlar o movimento da espada durante o ataque. Inicialmente, a colisão da *hitbox* da espada permanece desabilitada e só é ativada no momento exato do ataque. Ao final da animação, o próprio ***AnimationPlayer*** chama a função **queue_free()**, removendo a espada da cena automaticamente. A animação de ataque tem a opção ***Autoplay*** ativada, garantindo que todo o processo seja executado imediatamente após o objeto ser instanciado.

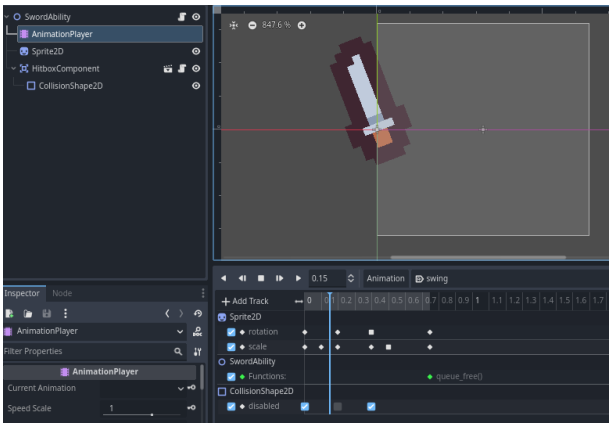

Figura 15. Animação da *SwordAbility*

Para gerenciar a habilidade, foi criado um controlador chamado ***SwordAbilityController*** (veja a Figura 16, responsável por definir o alcance, dano, frequência e melhorias da habilidade. O controlador instancia a espada próxima a um inimigo dentro do raio de ataque ao redor do jogador (linha 42). A frequência com que as espadas são criadas é controlada por um temporizador(linha 54), possibilitando que *upgrades* reduzam esse intervalo. O valor do dano é passado do controlador para a espada (linha 40), assim *upgrades* podem aumentar esse valor.

```
37    var sword_instance = sword_ability.instantiate() as SwordAbility
38    var foreground_layer = get_tree().get_first_node_in_group("foreground_layer")
39    foreground_layer.add_child(sword_instance)
40    sword_instance.hitbox_component.damage = base_damage * additional_damage_percent
41
42    sword_instance.global_position = enemies[0].global_position
43    sword_instance.global_position += Vector2.RIGHT.rotated(randf_range(0, TAU)) * 4
44
45    var enemy_direction = enemies[0].global_position - sword_instance.global_position
46    sword_instance.rotation = enemy_direction.angle()
```

Figura 16. *Script* do *SwordAbilityControler*

Na Figura 17, é possível observar um trecho do *script* do **SwordAbilityController**, especificamente a parte que controla os *upgrades*. Quando o jogador escolhe uma melhoria, um sinal é emitido, e o controlador escuta esse evento, verificando no dicionário de *upgrades* atuais se há melhorias correspondentes. Se houver, os ajustes são aplicados de acordo com a quantidade de *upgrades* acumulados.

```
49  func on_ability_upgrade_added(upgrade: AbilityUpgrade, current_upgrades: Dictionary):
50      if upgrade.id == "sword_rate":
51          var percent_reduction = current_upgrades["sword_rate"]\
52          ["quantity"] * .1
53          $Timer.wait_time = base_wait_time * (1 - percent_reduction)
54          $Timer.start()
55      elif upgrade.id == "sword_damage":
56          additional_damage_percent = 1 + (current_upgrades["sword_damage"]\
57          ["quantity"] * .15)
```

Figura 17. *Script* do *SwordAbilityControler*

Para permitir que o jogador adquira novas habilidades ou melhorias, foi necessário desenvolver um gerenciador de *upgrades*. A classe **UpgradeManager** faz o gerenciamento e

aplicação de habilidades e *upgrades* ao jogador.

Para controlar a probabilidade de cada habilidade ou melhoria ser oferecida ao jogador, foi utilizada uma tabela ponderada, na qual as habilidades e *upgrades* são armazenados com diferentes pesos de aparição. Dessa forma, habilidades mais raras têm menor chance de serem oferecidas, enquanto habilidades comuns aparecem com mais frequência.

Quando o jogador sobe de nível, um sinal é recebido e uma tela de seleção de *upgrades* é instanciada (linha 73), *upgrades* são escolhidos (linha 75), e são apresentadas (linha 76) ao jogador. Na Figura 18 é apresentado o trecho de código responsável por criar a tela de seleção de *upgrades*.

```
72 func on_level_up(current_level: int):
73     var upgrade_screen_instance = upgrade_screen_scene.instantiate()
74     add_child(upgrade_screen_instance)
75     var chosen_upgrades = pick_upgrades()
76     upgrade_screen_instance.set_ability_upgrades(chosen_upgrades \
77     as Array[AbilityUpgrade])
78     upgrade_screen_instance.upgrade_selected.connect\
79     (on_upgrade_selected)
```

Figura 18. Trecho de código da tela de seleção de *upgrades*

Após a escolha, a **UpgradeManager** processa a habilidade selecionada, verificando se ela já existe (linha 30). Caso a habilidade escolhida tenha atingido sua quantidade máxima (linha 38), ela é removida da tabela ponderada (linha 41), para que não seja oferecida novamente. A Figura 19 exibe a lógica da aplicação de habilidades.

```
28 func apply_upgrade(upgrade: AbilityUpgrade):
29     var  has_upgrade = current_upgrades.has(upgrade.id)
30     if !has_upgrade:
31         current_upgrades[upgrade.id] = {
32             "resource": upgrade,
33             "quantity": 1
34         }
35     else:
36         current_upgrades[upgrade.id]["quantity"] +=1
37 
38     if upgrade.max_quantity > 0:
39         var current_quantity = current_upgrades[upgrade.id]["quantity"]
40         if current_quantity == upgrade.max_quantity:
41             upgrade_pool.remove_item(upgrade)
42 
43     update_upgrade_pool(upgrade)
44     GameEvents.emit_ability_upgrade_added(upgrade, current_upgrades)
```

Figura 19. Lógica da aplicação de habilidades

Após esse processamento, a habilidade ou melhoria selecionada é aplicada ao jogador, garantindo sua evolução ao longo da partida. Esse sistema proporciona variedade na progressão do jogador e incentiva a tomada de decisões estratégicas na escolha das habilidades.

5.3.2. Telas e Menus

Foi criada uma tela de seleção de habilidades para exibir informações sobre as habilidades disponíveis e permitir que o jogador escolha qual deseja adquirir. As informações são organizadas em cartas, ilustrada na Figura 20, que possuem campos específicos para o nome da habilidade e sua descrição.

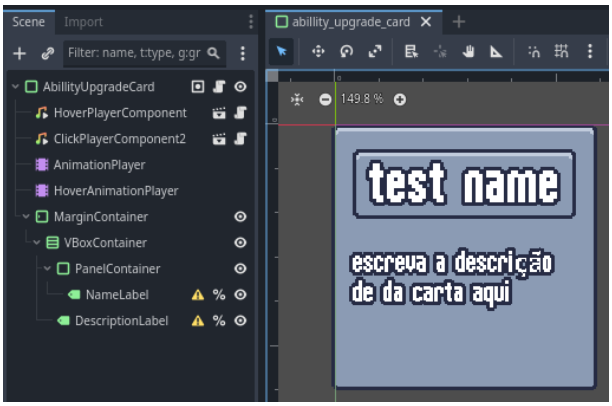

Figura 20. Carta de habilidade

As cartas possuem animações, que são acionadas nos seguintes momentos:

- **Animação "In"**: Quando a carta é criada.
- **Animação "Selected"**: Quando o jogador seleciona uma habilidade.
- **Animação "Discarded"**: Quando uma habilidade não é escolhida.
- **Animação "Hover"**: Quando o cursor do mouse passa sobre a carta.

Também foi desenvolvido um menu principal, visualizado na Figura 21, onde o jogador pode escolher entre:

- Iniciar uma nova partida.
- Comprar *upgrades* permanentes.
- Acessar o menu de opções.
- Sair do jogo.

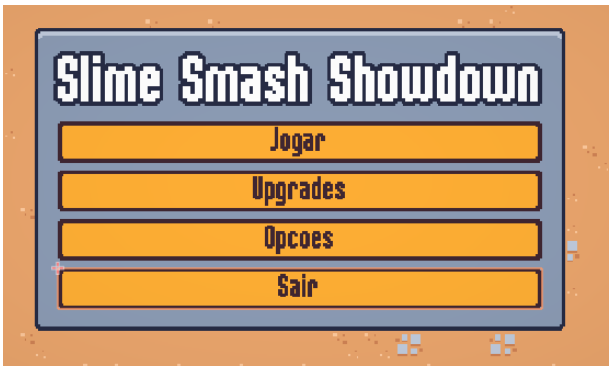

Figura 21. Menu principal

No menu de opções, visualizado na Figura 22 o jogador pode ajustar o volume dos efeitos sonoros e da música, além de alternar o modo de tela entre janela e tela cheia.

Figura 22. Menu principal

Já no menu de *upgrades*, visualizado na Figura 23, o jogador pode adquirir *upgrades* permanentes.

Figura 23. Menu de *upgrades*

As cartas de *upgrades* permanentes possuem os seguintes elementos:

- Nome do *upgrade*.
- Descrição do efeito.
- Quantidade máxima permitida.
- Preço em moeda do jogo.
- Barra de progresso, que indica quanto falta para o jogador conseguir comprar o *upgrade*.
- Botão de compra.

Os *upgrades* são organizados em um *layout* em grade, e caso a quantidade de cartas ultrapasse o número de colunas visíveis, um sistema de rolagem permite a navegação entre os itens disponíveis.

5.4. Testes e polimento

Com a base do jogo finalizada, iniciou-se a fase de testes para identificar problemas e ajustar o equilíbrio da jogabilidade.

Durante essa etapa, foi detectado um problema na geração de inimigos, onde alguns eram instanciados dentro da colisão das paredes, ficando presos. A causa do erro estava na verificação

da distância entre o jogador e a posição de criação, que não considerava o tamanho da área de colisão dos inimigos. Como consequência, eles podiam ser gerados muito próximos às paredes, impossibilitando seu movimento. Para corrigir essa falha, foi adicionada uma variável com um valor extra para aumentar o valor da distância a ser verificada, garantindo que os inimigos fossem criados apenas em posições válidas.

Os testes com jogadores, durante o trabalho da matéria de desenvolvimento de jogos, apontaram que a dificuldade não estava adequada, os feedbacks foram coletados pelo questionário do modelo EGameFlow. Para melhorar o equilíbrio do jogo, foram ajustados diversos parâmetros, como o dano e tempo de recarga das habilidades, a velocidade e a vida dos inimigos, além da frequência e quantidade de inimigos gerados. Essas mudanças tornaram o jogador mais forte e balancearam a geração dos inimigos, resultando em uma progressão mais justa e dinâmica.

Além dos ajustes na jogabilidade, diversas melhorias visuais foram implementadas para aprimorar a experiência do jogador, incluindo:

- Telas de transição entre menus, proporcionando uma navegação mais fluida;
- Efeito visual de impacto nos inimigos, aumentando o *feedback* dos ataques;
- Indicação visual de dano na tela sempre que o jogador for atingido;
- Ajuste nos *sprites* do jogador e dos inimigos, adequando-os à estética planejada;
- Adição de uma vinheta para reforçar a ambientação do jogo;
- Sistema de temas na *interface*, padronizando e estilizando os elementos visuais;
- Nova fonte, reforçando a identidade visual do jogo.

A parte sonora também recebeu ajustes significativos. Foram adicionados efeitos sonoros para os inimigos e o jogador ao serem atingidos, bem como sons para interações com a *interface*. Além disso, um *jingle* foi incluído para indicar a vitória ou derrota do jogador.

Para evitar repetições perceptíveis nos sons, foi desenvolvido um componente de randomização de *pitch*, que altera levemente a escala de reprodução dos efeitos sonoros a cada instância. Essa variação proporciona uma experiência auditiva mais rica e natural. A trilha sonora do jogo foi complementada com uma música de domínio público do *site* https://freepd.com/, garantindo um ambiente sonoro coeso e envolvente.

6. Conclusão

Este trabalho teve como objetivo o desenvolvimento do protótipo **Slime Smash Showdown**, um jogo *survivor-roguelike* criado na *engine* Godot, com foco em progressão dinâmica, combate, habilidades e geração procedural de inimigos. A concepção do projeto foi documentada em um GDD, que guiou a implementação gradual das mecânicas essenciais, incluindo movimentação, combate e progressão de habilidades.

Assim temos sistemas de combate, habilidades e progressão no qual o jogador consegue interagir para experimentar a interação lúdica, artificialidade por simular o combate e a sobrevivencia, uma disputa contra os inimigos, regras que definem o campo de batalha, a maneira como o jogador consegue enfrentar os inimigos e progredir e por fim um resultado com a vitoria ou derrota do jogador, todos esses elementos em conjunto se conectam com as ideias principais da definição de jogos proposta por [Salen and Zimmerman 2012], sistemas, jogadores, artificialidade, conflito, regras e resultado.

Para garantir escalabilidade e manutenção, adotou-se o padrão de composição. A modularização por componentes permitiu que diferentes elementos compartilhassem comportamentos sem dependência de herança rígida, facilitando a adição de novos inimigos e habilidades sem reescrita significativa de código.

O protótipo validou as mecânicas principais e possibilitou ajustes baseados em *feedback* dos jogadores. Testes indicaram que o jogo inicialmente apresentava uma dificuldade excessiva, levando a um balanceamento para tornar a experiência desafiadora, mas justa. Ainda há melhorias possíveis para dinamizar a progressão e a jogabilidade.

A escalabilidade do projeto foi uma prioridade, permitindo fácil expansão. O conhecimento adquirido em disciplinas como Linguagem de Programação, Programação Orientada a Objetos, Desenvolvimento de jogos, Engenharia de Software e Estrutura de Dados foi essencial para sua implementação estruturada.

Este projeto demonstra a viabilidade da Godot Engine no desenvolvimento de jogos independentes e a eficiência da abordagem baseada em componentes para facilitar manutenção e expansão. O jogo foi publicado no itch.io para validação pública e pode ser acessado em https://jellysoge.itch.io/slime-smash-showdown.

6.1. Trabalhos futuros

Para trabalhos futuros, diversas melhorias podem ser exploradas, como a introdução de novos inimigos e chefes com comportamentos mais complexos, a fim de elevar o nível de desafio e ampliar a diversidade dos combates. Adicionalmente, a implementação de novas habilidades e um sistema de árvore de habilidades permanentes permitirá aprofundar a customização e progressão dos personagens. Outras propostas incluem a adição de personagens jogáveis com estilos distintos e a criação de um *lobby* interativo, onde o jogador poderá acessar a árvore de habilidades, alternar entre personagens e interagir com NPCs[4] (Personagem não jogável, do inglês *Non-Player Character*). Essas evoluções visam tornar o jogo mais imersivo, variado e estratégico, proporcionando uma experiência mais rica e envolvente para os jogadores.

Referências

[Comando Geek 2024] Comando Geek (2024). O que é: Playtesting. Disponível em `https://comandogeek.com.br/blog/glossario/o-que-e-playtesting-2/`. Acessado em 09/03/2025.

[Costa 2019] Costa, T. A. (2019). Por que fazer a prototipação do seu software? Disponível em `https://blog.cronapp.io/prototipacao-de-software/`. Acessado em 09/03/2025.

[Devolver Digital 2016] Devolver Digital (2016). Enter the gungeon. Disponível em `https://enterthegungeon.com/`. Acessado em 03/09/2025.

[Fullerton 2008] Fullerton, T. (2008). *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*. Morgan Kaufmann.

[Galante 2022] Galante, L. (2022). Poncle. Disponível em `https://poncle.games/vampire-survivors`. Acessado em 26/02/2024.

[Holfeld 2023] Holfeld, J. (2023). On the relevance of the Godot Engine in the indie game development industry. Disponível em `http://arxiv.org/abs/2401.01909`. Acessado em 12/11/2024.

[Huizinga 2019] Huizinga, J. (2019). *Homo ludens: o jogo como elemento da cultura*. Editora Perspectiva, 1ª edition.

[Kalinin 2023] Kalinin, E. (2023). What are assets in game design? Disponível em `https://retrostylegames.com/blog/what-are-assets-in-game-design/`. Acessado em 09/03/2025.

[Kazoo 2019] Kazoo, S. (2019). Turnip boy commits tax evasion. Disponível em `https://snoozykazoo.com/games/turnip-boy-commits-tax-evasion`. Acessado em 26/02/2025.

[4]Qualquer personagem em um jogo que não é controlado pelo jogador, mas sim pelo próprio sistema. Eles podem ter funções diversas, como interagir com o jogador, oferecer missões, vender itens ou apenas compor o ambiente do jogo

[Linietsky et al. 2014] Linietsky, J., Manzur, A., and Godot community (2014). Godot docs – 4.3 branch. Disponível e mhttps://docs.godotengine.org/. Acessado em 26/02/2025.

[L'Italien 2024] L'Italien, R. (2024). Godot vs. the giants: What is godot and how it competes with unity and unreal. Disponível em https://www.perforce.com/blog/vcs/what-is-godot. Acessado em 26/02/2025.

[Lucchese and Ribeiro 2012] Lucchese, F. and Ribeiro, B. (2012). Conceituação de Jogos Digitais. *FEEC / Universidade Estadual de Campinas*, pages 1–9.

[Motion Twin 2018] Motion Twin (2018). Dead cells - the roguevania from motion twin. Disponível em https://dead-cells.com/. Acessado em 03/09/2025.

[Newzoo 2021] Newzoo (2021). Global games market report the vr & metaverse edition. Disponível em https://newzoo.com/insights/trend-reports/newzoo-global-games-market-report-2021-free-version/. Acessado em 05/11/2024.

[Petry 2020] Petry, L. C. (2020). O conceito ontológico de jogo. *Jogos digitais e aprendizagem: fundamentos para uma prática baseada em evidências*, pages 17–42.

[Rollings and Morris 2004] Rollings, A. and Morris, D. (2004). *Game Architecture and Design - A New Edition*. New Riders Publications.

[Salen and Zimmerman 2012] Salen, K. and Zimmerman, E. (2012). *Regras do jogo: fundamentos do design de jogos*. Blucher, 1ª edition.

[Schell 2008] Schell, J. (2008). *The Art of Game Design: A Book of Lenses*. CRC Press, 1 edition.

[Shu-Hui et al. 2018] Shu-Hui, C., Wann-Yih, W., and Dennison, J. (2018). Validation of EGameFlow: A self-report scale for measuring user experience in video game play. *Computers in Entertainment*, 16(3).

[Supergiant Games 2020] Supergiant Games (2020). Supergiant games. Disponível em https://www.supergiantgames.com/games/hades. Acessado em 10/01/2025.

A. Apêndice

Documento de Design de Jogo (GDD)
Autor: Gabriel da Silva Alves
Versão: 1.2

1. Visão geral

1.1. Gênero

- *Survivor-rogue-like*;
- Ação.

1.2. Plataforma

- PC;
- Steam.

1.3. Visão geral

Rogue-like de sobrevivência, onde *slimes* lutam contra hordas de heróis. A cada *Run*, o jogador ganha pontos que podem ser trocados por *upgrades* permanentes. O jogo contará com um *lobby* que pode ser aprimorado, concedendo bônus adicionais, similar ao sistema de Hades. Durante a *Run*, o jogador poderá escolher habilidades temporárias a cada nível evoluído, sendo que as opções disponíveis são aleatórias.

1.4. Controles

- **W, A, S, D**: movimentação;
- **Ponteiro do *mouse***: mira;
- **Clique do *mouse***: dispara;
- **Espaço**: *dash*;
- **TAB**: habilidade especial única do personagem.

2. Escopo do Projeto

2.1. Referências

- Vampire Survivors;
- 20 Minutes Till Dawn;
- TurnipBoy;
- Hades.

2.2. *Gameplay* resumo

O jogador enfrentará hordas de inimigos e poderá escolher poderes para ficar mais forte. Chefes surgirão ocasionalmente. Durante a *Run*, o jogador poderá coletar recursos para aprimorar o *lobby* e desbloquear melhorias, como um caldeirão da bruxa que permite evolução de habilidades.

2.3. Estágios

- **Arena:** Um campo aberto e infinito gerado proceduralmente, onde inimigos surgem continuamente e o jogador precisa sobreviver por um tempo determinado. Um modo com progressão por salas, similar ao do Hades, pode ser considerado.

3. Mecânicas de jogo

3.1. Mecânicas do jogador

- **Movimentação:** Visão de cima para baixo, permitindo movimentação em todas as direções;
- **Ataque básico:** Cada *slime* possui um ataque básico único, com diferentes efeitos e frequências de uso;
- **Habilidades especiais:** Cada *slime* possui uma habilidade única com tempo de recarga;
- **Experiência (XP):** Inimigos derrotados deixam XP, que ao atingir um valor específico, permite subir de nível;
- **Habilidades:** Ao subir de nível, o jogador pode escolher entre habilidades aleatórias;
- ***Upgrades*:** Durante a *Run*, o jogador pode obter pontos para *upgrades* permanentes;
- **Recursos:** Recursos coletados permitem melhorias no *lobby* e desbloqueio de bônus;
- ***Lobby*:** Após cada *Run*, o jogador retorna ao *lobby* principal para interações e melhorias.

3.2. Habilidades

- **Gato familiar:** Atira projéteis em inimigos;
- **Cobra familiar:** Persegue um inimigo e aplica veneno;
- **Morcego familiar:** Drena a vida dos inimigos e cura o jogador;
- **Espada mágica:** Ataca inimigos automaticamente;
- **Machado mágico:** Gira ao redor do jogador;
- **Tempestade:** Relâmpagos atingem inimigos aleatoriamente;
- **Poção de metamorfose:** Transforma temporariamente inimigos em sapos.

3.3. Mecânicas dos inimigos

- **Guerreiro básico**: Persegue e causa dano ao contato.
- **Arqueiro básico**: Mantém distância e dispara flechas.
- **Mago básico**: Dispara bolas de fogo de maior dano.
- **Lanceiro com montaria chefe**: Avança em alta velocidade contra o jogador.

3.4. Progressão

O jogador possui uma árvore de habilidades permanentes no caldeirão da bruxa. Durante a *Run*, evolui de nível e adquire poderes. Chefes concedem poderes especiais e inimigos mais fortes surgem com o tempo.

4. História e personagens

4.1. Enredo

Os humanos caçam os *slimes* para usar seus restos em armas e energia mágica. O protagonista, prestes a ser derrotado, é salvo por uma bruxa que precisa dos *slimes* para experimentos. Ela lança um feitiço que o teletransporta ao *lobby* sempre que quase morre (mecânica *rogue-like*). Com o tempo, o *slime* recruta aliados para lutar contra os humanos.

4.2. Personagens principais

- **Jelly (nome provisório):** *slime* com uma planta na cabeça, usa uma *glock*;
- **Codessa (nome provisório):** Bruxa (robô) que auxilia os *slimes*, mas possui segundas intenções;
- **Steelon (nome provisório):** *slime* dentro de uma armadura, usa uma espada;
- **Smashia (nome provisório):** *slime* feminina com manoplas mágicas.

4.3. Missões e objetivos

Ajudar a bruxa, impedir a extinção dos *Slimes* e derrotar os humanos.

4.4. Ficha técnica dos personagens

- **Jelly**: Atira uma bala perfurante e crítica.
- **Steelon**: Ataque giratório, ganha força temporária.
- **Smashia**: Golpeia rapidamente, ganha aumento de velocidade.

5. Aspectos técnicos

5.1. Ferramentas e tecnologias

- Godot;
- Aseprite.

5.2. Requisitos de sistema

- Microsoft Windows.

Documento Digitalizado Restrito

Versão final do artigo do TCC do aluno GSA

Assunto: Versão final do artigo do TCC do aluno GSA
Assinado por: Isaias Oliveira
Tipo do Documento: Comprovante
Situação: Finalizado
Nível de Acesso: Restrito
Hipótese Legal: Informação Pessoal - dados pessoais e dados pessoais sensíveis (Art. 31 da Lei nº 12.527/2011)
Tipo do Conferência: Documento Digital

Documento assinado eletronicamente por:

- **Isaias Mendes de Oliveira**, **PROFESSOR ENS BASICO TECN TECNOLOGICO**, em 11/03/2025 08:10:34.

Este documento foi armazenado no SUAP em 11/03/2025. Para comprovar sua integridade, faça a leitura do QRCode ao lado ou acesse https://suap.ifsp.edu.br/verificar-documento-externo/ e forneça os dados abaixo:

Código Verificador: 1961284
Código de Autenticação: b7c8cf4b83